EECS 351-1: Intro to Computer Graphics — Proj C: Better Lights & Materials

Instructor: Jack Tumblin — **Winter 2019**

Your mission in Project C is to create realistic interactive lighting and materials in WebGL in a 'virtual world.' As before, users can move and explore 3D animated solid objects placed on a patterned 'floor' plane that stretches to the horizon in the x, y directions. Unlike Project B, the objects in this 'virtual world' are made from different materials, each with individually-specified emissive, ambient, diffuse, specular parameters. The world also contains several smoothly-movable user-adjustable light sources, each with different position, ambient, diffuse, and specular parameters. Your program then uses lights, materials, surface-normals and more with several vertex/fragment shader pairs to compute the Phong lighting model in different ways, including 'Gouraud' shading (yields faceted appearance) and 'Phong' shading (for smooth-looking, facet-free surfaces with nicely rounded specular highlights).

Requirements: **Project Demo Day (and due date): Mon Mar 11, 2019**

A)-- In-Class Demo: As with Projects A & B, on the due date (Mon Mar 11) you demonstrate your completed program to the class. Two other students evaluate your work on a 'Grading Sheet', and we all share our best project ideas. Based on Demo Day advice, you then have ~72 hours to revise and improve your project before submitting the final version for grading. Your project grade is then determined from your final version.

B) --Submit your finalized project to CMS/Canvas no more than 72 hours later(Thurs Mar 14, 11:59PM) to avoid late penalties. Submit just one single compressed folder (ZIP file) that contains:
1) your written project report as a PDF file, and
2) one folder that holds sub-folders with all JavaScript source code, libraries, HTML, etc. (mimic the 'starter code' ZIP-file organization). We must be able to read your report & run your program in the Chrome or Firefox browser by simply uncompressing your ZIP file, running an HTML file found inside, in the same directory as your project report.
---**IMPORTANT:** Name your ZIP file and the directory inside as: **Familyname**Personalname_ProjC
 For example, my project C file would be: TumblinJack_ProjC.zip. It would contain sub-directories such as 'lib' and files such as TumblinJack_ProjC.pdf (a report), TumblinJack_ProjC.html, TumblinJack_ProjC.js ,etc.
---To submit your work, upload your ZIP file to Canvas→Assignments. DO NOT e-mail! (rejects executables).
---BEWARE! SEVERE LATE PENALTIES! (see Canvas→Assignments, or the Syllabus/Schedule).

Project C consists of:
1)—Report: A short written, illustrated report, submitted as a printable PDF file.
Length: >1 page, and typically <5 pages, but you should decide how much is sufficient.
A complete report consists of these 3 sections:

a)--your name, netID, and a descriptive title for your project
(e.g. Project C: Jeweled Starfish and Crabs Scuttle on Sparkling Sand, not just "Project C")

b)—a brief 'User's Guide'. Begin with a paragraph that explains your goals, then give user instructions on how to run and control the project. (*e.g.* "Arrow keys aim camera (yaw,tile WASD keys move/strafe camera; Q/E keys rotate inner ring forwards/backwards; HUD text for range (Km)") Your classmates should be able to read ONLY this report and easily run and understand your project without your help.

c)—a brief, illustrated 'Results' section that shows **at least 4 still pictures** of your program in action (use screen captures; no need for video capture or gifs), with figure captions and text explanations. Your figure(s) also must include a correctly-drawn sketch of your program's scene graph (the 'tree of transformations': unsure? See lecture notes, such as:
2018.01.22.VectorMathPart2_DualitySceneGraphs_VanDamm05).

2)—Your Complete WebGL Program, which must include:
a)---User Instructions: When your program runs, it must explain itself to users. How? You decide! Perhaps

print a brief set of user instructions below the canvas object? Or print 'press F1 for help'? Create a pop-up window? Perhaps within the 'canvas' element using the 'HUD' method in the book, or in the JavaScript 'console' window (in Google 'Chrome' browser), etc. Your program should never puzzle its users, or require your presence to explain, find, or use any of its features.

b)---'Ground Grid' Surface: Your program must clearly depict a horizontal 'ground grid' that extends to the horizon: a very large, repetitious pattern of repeated lines, triangles, or any other shape that repeats to form a vast fixed 'floor' of your 3D world, a detailed visual pattern that helps users sense camera aiming and movement. The ground must span the **x,y plane** (z= ~0) of your 'world-space' coordinate system; *do not* use +y' as 'up'; use +z! This ground should make any and all camera movements obvious on-screen, and form a reliable 'horizon line' when viewed with a perspective camera. HINT: you can make plausible terrain from a ground plane made of triangles if you a) assign modestly randomized terrain-like materials at each vertex, and b) displace the vertices by +/-z with the sum of a few 2D sine-waves at different non-harmonic wavelengths, or by fractal/subdivision methods (See: **http://www.gameprogrammer.com/fractal.html**) .

c)---At least 3 solid (not wireframe) separately-located, jointed, 3D animated objects with sensible surface normals at each vertex (otherwise your Phong Lighting results look strange/wrong). These jointed objects must change their joint angles continually and smoothly, without requiring any user input to continue moving. For example, could you make a tree that waves in the wind (from cylinders)? Place each jointed object at a different location on the 'ground plane'. You may re-use Project A & B shapes, but must 'light' them, which will require each vertex to include its own, separately-specified surface-normal vector attribute.

d)---Scene must contain at least one large, slowly-spinning sphere at a fixed location in the scene that we can view and light from any direction to let users visually confirm that all forms of shading (e.g. Phong, Gouraud, flat…) and lighting (e.g. Phong, Blinn-Phong, Cook-Torrance, etc.) work correctly.

e)---A displayed result in a single viewport that always completely fills a user-resizable window, but never over-fills it. No matter how tall or how wide the window, your code must fill the entire window with the output of a perspective camera whose vertical field of view is always 30 degrees from top edge to bottom edge, and whose aspect ratio matches the display window. While you may add a fixed-height window region to hold HTML buttons & text; otherwise, your window cannot contain any variable-sized 'blank' areas, and re-sizing the window must not distort any displayed objects: circles must remain circles for any window height and any window width. CAUTION: your program should never cause the browser to create 'scroll bars': these appear when your HTML web-page uses an area LARGER than the newly re-sized browser window can display.

f) —View Control: smoothly & independently control 3D Camera positions and aiming direction. Both, together! Your code must enable users to explore the 3D scene via user interaction. I recommend that you use arrow keys, W/A/S/D, mouse-dragging, or other widely-used key combinations to steer and move through the scene. You may design and use your own camera-movement system, but for full credit your system must allow complete 3D freedom of movement:

1. at any 3D location, your camera MUST be able to smoothly pivot its viewing direction without any change in 3D position. (I suggest: arrow keys to tilt/yaw camera; up vector 0,0,1). If you pretend that your head is the camera, you must be able to turn your head without moving your body), and:
2. your camera MUST be able to move to any 3D location from any other 3D location in one straight line, WITHOUT changing its viewing direction as it moves. You MUST NOT require users to move in only the x,y,z directions, or only in circles of varying radius! (I strongly recommend: W/S keys to move forward/back in viewing direction, and A/D keys 'strafe' left/ right: move horizontally, perpendicular to viewing direction).
   For example: imagine a scene of 64 colorful cubes placed in a 4x4x4 grid above the 'ground plane' to form a city of floating buildings and flying cars (e.g. **http://youtu.be/IJhlD6q71YA?t=29s**). However, these streets don't follow the x,y,z directions –the 4x4x4 grid was rotated to place two opposite corners on the z axis: its streets align with vectors (1,1,1) , (-1,1,1), and (1,-1,1), and the

cube-of-streets slowly tumbles; it rotates at 30 degrees/hour around an axis whose orientation also changes very slowly). Your camera must be able to'drive' down those streets easily; thread itself through all the streets and around the irregular grid of buildings, moving smoothly without any awkward zig-zagging. If your system cannot easily position the camera to 'drive around the block' in 3D, if your system rotates the camera when users move the camera, or moves the camera when users rotate the camera then it does not meet the project requirements.

BIG HINTS: If you use `LookAt()` to create your 'view' matrix, your user controls must modify BOTH the camera position (VRP or 'eye') AND the target point or 'look-at' point, and vary them independently. In class we described the 'glass-cylinder' model for camera movement that easily achieves all the Project B goals.

g)---Assign obviously different-looking Phong materials to each object shown on-screen.
Each of your 3 (or more) objects must use (or select) a different visually-distinct material, an easy-to-access set of **ALL 13** parameters or more that describes material response to light in the Phong lighting model. The parameters are: 9 floating-point color-reflectance values $0.0 \leq R,G,B \leq 1.0$, for ambient, diffuse, and specular reflectance (K_a, K_d, K_s); 3 floating-point color-emittance values K_e ($0.0 \leq R,G,B \leq 1.0$, but usually zero; these are the 'glow-in-the-dark' values for the material), and a 'shinyness' coefficient that sets the size of the specular highlight seen on a surface: smaller highlight$\rightarrow$larger shininess component $n_{shiny.}$ (Starter code helps!)

h)---Create at least two point-like, non-directional light sources that will illuminate your objects using the Phong lighting model. Keep one light fixed to the camera position (a 'headlight'), and place the other light in 'world' coordinates, at user-adjustable 3D position. Your program probably needs to make a light-describing object for each light: an easy-to-access set of parameters that describe how the light source will affect the appearance of nearby materials, including: light-source 3D position coordinates, and Phong light-source strengths ($0.0 \leq R,G,B \leq 1.0$) for ambient, diffuse, and specular illumination (I_a, I_d, I_s).

--For this project, each light source must allow users to switch on/off each light-source component independently and separately (*e.g.* ambient light on/off, diffuse light on/off, specular light on/off).

--For this project, you may safely ignore light attenuation by the distances between the light source and each illuminated surface; light source position determines only the direction from the light source to a point on a surface, and ***not*** the incident illumination intensity. As you will find, $1/r^2$ attenuation often looks 'too dark'. NOTE: If you implement distance dependent attenuation, give users easy ways to adjust or disable it!

j)---Write your own vertex shaders and fragment shaders to implement ≥4 selectable shading methods.
Your program must allow users to switch between these lighting and shading methods interactively (via keyboard, mouse, or HTML buttons, etc.), without stopping or disrupting the program or it's on-screen display. Users must be able to select between at least these 4 methods:

a) Phong lighting with Phong Shading, (no half-angles; uses true reflection angle)
b) Blinn-Phong lighting with Phong Shading (requires 'half-angle', not reflection angle)
c) Phong lighting with Gouraud Shading (computes colors per vertex; interpolates color only)
d) Blinn-Phong lighting with Gouraud Shading (computes colors per vertex; interpolates color only)

Combining the Phong lighting model (ambient, diffuse, specular, emissive) with Phong or Blinn-Phong shading dramatically improves the appearance of realism and smooth surfaces in OpenGL/WebGL, because the specular highlights are round, move smoothly, and do not have the faceted appearance of Gouraud Shading. By selecting between those methods using an interactive 'uniform' variable, users can change shading/lighting as their program runs and see exactly how they differ on-screen.
Your GLSL Phong *shading* program must interpolate surface normals, and possibly other vectors needed for lighting. It will supply them to your Fragment Shader via GLSL 'varying' variables, and will use these per-pixel normal values to compute pixel colors from vector calculations such as dot products. This method dramatically improves the quality of specular highlights; the surface appears smooth and the highlights appear rounded instead of the faceted highlights of Gouraud shading.

k)—EXTRA CREDIT: GLSL Geometry Distortions.
Nonlinear shape adjustments in Vertex Shader (possibly animated)
that no matrix transform can duplicate.

You already know at least two different distortion methods: change vertex positions as a function of vertex position; change surface normal as a function of vertex position; make at least one of them time-varying. For example, you may 'twist' vertices around an object's own z-axis by applying a different z-axis rotation to each vertex. The shader can accept objects' vertices in their unmodified 'model' or 'local' coordinates, and rotate each one around the z axis by the z-dependent amount (z*twist) degrees before applying your own version of the model and view matrix.

Or perhaps you want to impart a local 'wavyness' to 3D space? You could make a 3D sinusoidal displacement field: a function of x,y,z that provides a value between -1 and +1 that varies smoothly with position. Evaluate the displacement field at the position of the current vertex, and add it to the vertex's position.

Better yet, devise your own local, time-varying nonlinear spatial distortions. **NOTE**: by 'local' geometry, I mean these distortions must be applied in the coordinate system axes of the individual objects, and not in the shared 'world', 'eye' or other coordinates. The distortions must not change when you move an object to a different 3D location, and they must not change as you move or aim the camera differently.

l)---EXTRA CREDIT: Texture Mapping.

Apply image textures to one or more objects in your scene. Please note that this DOES NOT replace the requirements for Phong-lit materials for this project; you must still meet all of those requirements as well. Note that online, in our Lengyel reading, and in the latter parts of our WebGL book you can find information on 'bump mapping', a commonly used method in modern computer games to apply image values as displacements to local surface normals. This method greatly boosts the visual complexity and richness of shapes without a high vertex count. You can also find information on render-buffer and texture buffer operations such as 'render-to-texture' that permit you to render mirrors that show other views of the scene. Try it!

Sources & Plagiarism Rules:

Simple: *never* submit the work of others as your own.

You are welcome to begin with the book's example code and the 'starter code' I supply; you can keep or modify any of it as you wish without citing its source. I strongly encourage you to always start with a basic graphics program (hence 'starter code') that already works correctly, and incrementally improve it; test, correct, and save a new version at each step.

I ***want*** you to explore -- to learn from websites, tutorials and friends anywhere (e.g. GitHub, StackOverflow, MDN, CodeAcademy, OpenGL.org, etc), and to apply what you learn in your Projects.
Please share what you find with other students, too -- list the URLs on CMS/Canvas discussion board, etc. and list in the comments the sources of ideas that helped you write your code.

BUT always, ALWAYS credit the works of others— ***no plagiarism!***

Plagiarism rules for writing essays apply equally well to writing software. You would never cut-and-paste paragraphs or whole sentences written by others and submit it as your own writing: and the same is true for whole functions, blocks and statements. ***Take their good ideas, but not their code*** add a gracious comment that recommends the inspiring source of those good ideas, and then write your own, better code in your own better style; stay compact, yet complete, create an easy-to-read, easy-to-understand style.
Don't waste time trying to disguise plagiarized code by rearrangement and renaming (MOSS won't be fooled).
Instead, study good code to grasp its best ideas, learn them, and make your own version in your own style.
Take the ideas alone, not the code: make sure your comments properly name your sources.

Also, please note that I apply the 'MOSS' system from Stanford (https://theory.stanford.edu/~aiken/moss/) and if I find any plagiarism evidence (sigh), the University requires me to report it to the Dean of Students for investigation. It's a defeat for all involved: when they find misconduct, they respond in very strict and very punitive ways.